

GO-02 05: Основы языка. Конструкции языка и функции

Описание:

Конструкции языка

В этом разделе вы познакомитесь с конструкциями языка, которые позволяют управлять потоком выполнения программы, и напишите первые функции. Начнем с условного оператора.

Условный оператор

Синтаксис условного оператора if следующий:

```
if logicValue {  
    fmt.Println("Value is true")  
} else {  
    fmt.Println("Value if false")  
}
```

Оператор проверяет условие на истинность и выполняет тот или иной блок кода в зависимости от написанной конструкции.

В качестве logicValue может быть только значение логического типа, в отличие от языков, в которых реализовано приведение типов, где 0 или пустая строка приводится к значению false. При попытке это сделать в Go вы увидите ошибку.

```
logicValue := 0  
if logicValue {      // non-bool logicValue (type int) used as if  
condition  
    fmt.Println("Value is true")  
}
```

Логическому выражению в операторе if может предшествовать выражение присвоения с использованием сокращенного оператора присваивания :=. В этом случае можно получить какое-то значение и тут же его использовать в логическом выражении. Это бывает полезно, например, при проверке ключа на существование.

```
customer := map[string]string{  
    "name": "John",  
    "lastName": "Smith"}  
  
if lastName, isExist := customer["lastName"]; isExist {  
    fmt.Printf("Hello Mr. %s!\n", lastName)  
}
```

В языке Go поддержание чистоты кода лежит в зоне ответственности самого языка, и если вы, например, не используете какие-то объявленные переменные, то увидите ошибку на этапе компиляции программы. Поэтому если какие-то из переменных в выражении не будут использованы, то их можно заменить пустым оператором `_`. Изменим код примера и выведем сообщение, если ключа не существует.

```
if _, isExist := customer["lastName"]; !isExist {  
    fmt.Println("Empty field!")  
}
```

Все переменные, которые были объявлены в блоке инициализации, доступны в любой ветви оператора `if`, но не за его пределами. Поэтому если нужны какие-то локальные переменные для использования в логическом выражении оператора, то блок инициализации отлично для этого подходит.

```
if lastName, lastNameExist := customer["lastName"]; !lastNameExist {  
    fmt.Println("Empty lastName!")  
} else if name, nameExist := customer["name"]; !nameExist {  
    fmt.Println("Empty name")  
} else {  
    fmt.Printf("Hello, %s %s!\n", name, lastName)  
}
```

Стоит упомянуть о тернарном операторе, к которому вы, может быть, привыкли при работе с другими языками программирования. В Go этот оператор не реализован. Его нет, так как, по мнению разработчиков языка, он часто используется для построения сложных и длинных выражений, которые неочевидны и трудночитаемы. Конструкция `if ... else` в этом плане более прозрачная и потому предпочтительней, хоть и, казалось бы, многословней.

Оператор выбора `switch`

Бесконечно писать конструкции из блоков `else if` не является хорошей практикой, поэтому когда вариантов выбора больше, чем два, то наиболее подходящим вариантом будет оператор `switch`. Оператор `switch` проверяет переменную на соответствие одному из вариантов, описанных в блоках `case`, и выполняет код, расположенный в выбранном блоке `case`.

```
userRole := "admin"  
switch userRole {  
case "admin":  
    fmt.Println("Access granted")  
case "user":  
    fmt.Println("Access granted")  
default:  
    fmt.Println("Access denied")
```

```
}
```

На примере выше указан еще один необязательный блок `default`, который будет выполнен, если для значения переменной `userRole` не будет найдено соответствий.

Оператор `switch` тоже поддерживает блок инициализации переменной, который через точку с запятой может быть добавлен перед проверяемой переменной.

В других языках программирования, как правило, после выполнения кода из какого-либо блока `case` происходит последовательное выполнение кода из последующих блоков, если не указано ключевое слово `break`. Конечно, иногда это бывает полезным, например, если для двух разных значений должен выполняться один и тот же код. Но чаще приходится писать `break`, чтобы выйти из блока `switch`.

В этом плане алгоритм работы оператора `switch` отличается в Go. В блоке `case` через запятую можно указать несколько значений, что позволит выполнить один и тот же код в разных случаях. Но если в блоке `case` не указано ключевое слово `fallthrough`, то по завершению обработки выбранного блока будет выполнен выход из оператора `switch`.

```
userRole := "guest"
switch userRole {
case "admin", "user":
    fmt.Println("Access granted")
case "guest":
    fallthrough
default:
    fmt.Println("Access denied")
}
```

В синтаксисе `switch/case` ключевое слово `break` применяется в том случае, если надо прервать выполнение текущего блока `case`.

Оператор `switch` может быть составлен без переменной, тогда это эквивалентно записи `switch true {...}`. В этом случае в блоках `case` должны быть составлены логические выражения с участием необходимых переменных.

Циклы

В Go есть разные циклы, но их синтаксис немного отличается от того, что вы привыкли видеть в других языках программирования. В Go только одна инструкция, при помощи которой реализуются все виды циклических алгоритмов, — это инструкция `for`. В классическом варианте это обычный цикл, который имеет три выражения, расположенных последовательно через точку с запятой: выражение инициализации переменных, логическое выражение, которое выполняется в начале каждой итерации, и выражение, которое выполняется в конце каждой итерации.

```
for i := 0; i <= 10; i++ {
    fmt.Println(i)
}
```

Все три выражения не являются обязательными — таким образом и получаются различные виды циклов.

Бесконечный цикл реализуется при помощи выражения `for` или `for ;;;`, прервать работу цикла можно при помощи ключевого слова `break`. Пропустить дальнейшее выполнение кода и перейти к следующей итерации можно при помощи ключевого слова `continue`.

```
var isBreak = true
for {
    if isBreak {
        break
    }
}
```

Если конструкция состоит из нескольких вложенных циклов и при каком-то условии необходимо выйти из них, то можно использовать метки. Для этого надо расставить метки в нужных местах и указать имя метки в качестве операнда для `break`.

```
isBreak := true
Loop1:
    for {
        fmt.Println("Loop 1")
        for {
            fmt.Println("Loop2")
            if isBreak {
                fmt.Println("break")
                break Loop1
            }
        }
    }
    fmt.Println("After Loop")
```

Цикл с условием, аналогичный `while` в других языках, реализуется при помощи конструкции `for` выражение.

```
iterate := true
for iterate {
    fmt.Println("Iteration")
    iterate = false
}
```

И, наконец, наверное, один из самых часто используемых видов циклов, - это цикл по диапазону, который позволяет удобно перебрать слайс или мапу без многословных конструкций вышеописанных циклов, когда надо самостоятельно проинициализировать значения переменных, выполнять инкремент и т.д. Цикл по диапазону реализуется при

помощи конструкции `for ... range {}`. Как вы уже знаете из предыдущего раздела, оператор `range` возвращает два значения: ключ и значение текущего элемента. Если вам нужны только значения, то необязательное значение ключа можно пропустить при помощи пустого оператора `_`. В другом случае, если вы итерируетесь по ключам, можно указать только одну переменную.

```
m := map[string]string{
    "key1": "val1",
    "key2": "val2",
}
for k, v := range m {
    fmt.Println(k, v)
}
for k := range m {
    fmt.Println(k)
}
for _, v := range m {
    fmt.Println(v)
}
```

Функции

Объявление и возвращаемые значения

Обычная функция в Go выглядит примерно следующим образом:

```
func funcName(param int) int {
    param++
    return param
}
```

Объявление функции начинается с ключевого слова `func`, после чего следует название самой функции. В круглых скобках описываются принимаемые аргументы с указанием типов. Завершает объявление указание типа возвращаемого функцией значения.

Если функция принимает несколько аргументов одного типа, то тип можно указать один раз, перечислив все аргументы через запятую.

```
func funcName(param1, param2, param3 int) int{
    return param1 + param2 + param3
}
```

Функции в Go могут возвращать больше, чем одно значение. Вы часто будете встречать функции, которые возвращают, как правило, первым значением результат работы, а вторым — ошибку. Об ошибках и способах их обработки вы узнаете позже, а сейчас достаточно узнать, что типы нескольких возвращаемых значений можно указать в скобках через запятую.

```
func foo(p1 int, p2 string) (int, string) {
```

```
    return p1, p2
}
```

Кроме этого, Go позволяет объявлять именованные возвращаемые значения. Для этого перед типом возвращаемого значения следует указать название.

```
func sum(p1 int, p2 int) (sum int) {
    sum = p1 + p2
    return
}
```

Именованные возвращаемые значения инициализируются значениями по умолчанию указанных типов. То есть, в момент выполнения функции возвращаемый результат уже имеет какое-то значение, поэтому оператор `return` можно использовать без операндов.

Вариативные функции

В языке реализована возможность передавать в функции переменное число параметров. Для этого используется оператор `...`. Синтаксически объявление вариативной функции выглядит следующим образом:

```
func words(w ...string) []string {
    return w
}
```

Оператор `...` указывается перед типом данных и упаковывает все параметры указанного типа в слайс элементов этого же типа — тип возвращаемого значения об этом явно говорит. Распаковать слайс, чтобы элементы можно было передать в качестве параметров в такую же вариативную функцию, можно тоже при помощи оператора `...`, который указывается после переменной.

```
func words(w ...string) []string {
    fmt.Println(printWord(w...))
    return w
}
```

```
func printWord(w ...string) int {
    for _, word := range w {
        fmt.Println(word)
    }
    return len(w)
}
```

Анонимные функции и замыкания

В языке Go реализованы анонимные функции. Их применение позволяет писать более гибкий код. Анонимные функции являются неименованными и могут быть определены практически в любом месте вашего кода. Анонимную функцию можно объявить и тут же

вызвать, передав нужные параметры, или подменять реализацию в зависимости от каких-либо условий и присваивать в переменную, чтобы вызвать в нужный момент. На примере ниже показан уже знакомый вам код, только он переписан с применением анонимных функций.

```
printWord := func(w ...string) {  
    for _, word := range w {  
        fmt.Println(word)  
    }  
}
```

```
func (w ...string) {  
    printWord(w...)  
}("str1", "str2")
```

Когда вы начинаете часто применять анонимные функции в программе, возникает необходимость типизации их сигнатуры. Go предоставляет такую возможность. Можно объявить именованный тип функции при помощи ключевого слова `type`. Далее этот тип можно указывать в функциях в качестве типа принимаемого значения и передавать в них функцию как аргумент. Немного изменим наш пример.

```
type wordLogger func(w ...string) (cnt int)
```

```
func main(){  
    printWord := func(w ...string) (cnt int) {  
        for _, word := range w {  
            fmt.Println(word)  
        }  
  
        return len(w)  
    }  
  
    func (printer wordLogger, w ...string) {  
        fmt.Printf("Count words: %d\n", printer(w...))  
    }(printWord, "str1", "str2")  
}
```

Теперь узнаем, как работать с замыканиями в Go. Замыкания — это такой тип анонимных функций, которые обращаются к переменным, объявленным за пределами блока функции, но в то же время находящимся в области видимости функции.

```
func main() {  
    sequence := sequenceGen(4)  
    fmt.Println(sequence())    // 4
```

```

fmt.Println(sequence()) // 8
fmt.Println(sequence()) // 12
fmt.Println(sequence()) // 16
fmt.Println(sequence()) // 20
fmt.Println(sequence()) // 24
fmt.Println(sequence()) // 28
fmt.Println(sequence()) // 32
}

func sequenceGen(n int) func() int {
    i := 0
    return func() int {
        i += n
        return i
    }
}

```

Полезные ссылки:

- [The anatomy of Functions in Go](#)
- [Understanding Function Closures in Go](#)
- [5 Useful Ways to Use Closures in Go](#)

Задание:

1. Создайте в проекте module02 новую ветку 05_task.
2. Создайте новую директорию с файлом main.go.
3. Напишите функцию contains, которая принимает на вход два параметра: слайс строк a и строку x. Функция должна проверять, содержится ли строка x в слайсе a, и возвращать булево значение.
4. Создайте вариативную функцию getMax, которая находит максимальное целое число из переданных на вход параметров.
5. Выведите на экран результат вызова функций.
6. Создайте коммит с вашим решением и отправьте ветку удаленный репозиторий.
7. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки 05_task.